Quadruped Robot Locomotion with Reinforcement Learning
OpenAI Gym Code and Report (C&R)

GROUP 18: HO KAE BOON (A0219811H), HUANG WENBO (A0285085W), ZENG ZHENGTAO (A0285134E), ZHONG HAO (A0284914W)

Our project aims to train a quadruped robot locomotion using reinforcement learning. The first step in this project is to set up an OpenAI Gym environment. To achieve this, we primarily utilize the PyBullet library, which is a lightweight physics simulation designed for robot dynamics simulation and provides an excellent platform to build our quadruped locomotion simulation. The quadruped robot employed in this project is Unitree's Aliengo. Our environment is single-agent, static, deterministic, partially observable with continuous state. Meanwhile, our action space is discrete.

In the 'aliengo_gym.py' file, we define a 'SAPPEnv' class that inherits from the 'Env' class from OpenAI Gym. 'SAPPEnv' initialization will establish a connection with the PyBullet physics simulation engine during initialization. By default, we keep the Graphical User Interface (GUI) of PyBullet hidden to be more resource efficient, but it can be displayed by providing the 'render=True' argument, to view the background simulation process. The initialized PyBullet environment starts empty and devoid of any element. The 'SAPPEnv' class encapsulates several methods that govern the interactions with the environment. We have provided the demo code inside the 'demo_code.ipynb' file for you to try them out. The more relevant methods include:

reset():
• Used to reset the PyBullet simulation and gym environment.
• When running *reset()* for the first time, we need to provide an argument, 'reload_urdf=True'. This will configure PyBullet properties such as the timestep, gravity, and built-in camera position. It will also import the robot model and floor model with their initial kinematic state into the empty PyBullet with the help of the Universal Robot Description Format (URDF) files for both the Aliengo model and floor model found inside the 'pybullet_data' folder. The URDF file specifies information about links, joints, as well as visual, collision, inertial, material and transmission properties of the model, to closely simulate a real-life Aliengo quadruped.
• For subsequent *reset()* run, to start an episode, there's no need to reimport the model. We only need to reset the existing robot models and the built-in camera back to their initial position and kinematic state.
• After resetting the robot's kinematic state, we would then update the value for state, robot observation, and the next valid state variables.

get_state():
• Used to return the current state information (dictionary format), including:
1) "robot_torso_position": # px, py, pz values w.r.t. pybullet origin frame.
2) "robot_torso_velocity": # vx, vy, vz values w.r.t pybullet origin frame.
3) "robot_torso_orientation": # quaternion values w.r.t. pybullet origin frame.
4) "each_joint_angle": # current angle for our focused eight joints.
5) "any_collision_between_each_link": # Boolean → if True, the current state is Termination State.
• We use PyBullet functions to directly extract the torso link and each joint's information from the simulation. For collision detection, we iterate through every two link combinations and use PyBullet functions to check for any contact points.

get_observation():
• Used to return the agent's observation (dictionary format) of the current state.
• To align with real-world robots, the observation consists of proprioceptive information that can be obtained from robot sensors, including IMU data, odometry data, motor encoder data, etc. This

information includes every state variable except “any_collision_between_each_link”. Consequently, this method reads the value directly from the current state.

listNextValidActions():
• Used to return the valid actions (list format, discretized) for the current state.
• The joint motors are velocity-controlled. The output consists of a list of tuples, where each tuple contains 8 values corresponding to the (FR_upper, FR_lower, FL_upper, FL_lower, RR_upper, RR_lower, RL_upper, RL_lower) joint control value. This control value can be -10, -1, +1 or +10, defined such that, +/- sign of the control value is the velocity (+1 or -1 rad/s), while magnitude of the control value is the maximum torque (Nm) allowed. Since PyBullet will automatically stop the rotation of joints that exceeded their angle limit (specified in the URDF file), we will ignore those cases and set the valid actions to constant for all states. After calculating the combination of 8 joint values, we return a list of length 4^8 = 65536 actions.

step(action):
• Used to execute the provided action for a single time step and return the following:
1) next state, 2) reward, 3) Whether it is a termination state (collision between each link occurs), and 4) next state’s all valid actions.
• We iterate through the eight joints, applying the corresponding torque values using PyBullet functions. Subsequently, we enable the PyBullet simulation to run for a specified timestep. Following this, we retrieve the new state, calculate the reward (as explained in the next point), obtain the next valid actions, and then return all these values.
• Reward calculation involves comparing state values to a threshold value predefined in the code. The final reward is determined by summing the following penalties/rewards:
A velocity reward (+20) when robot linear velocity in the targeted direction > threshold;
A deviation penalty (-5) when robot sideways velocity > threshold;
A height penalty (-5) when robot torso height < threshold;
A pitch penalty (-5) when the robot torso has a large pitch angle > threshold;
A large-angle penalty (-2) when robot joint angle > threshold.

render():
• Used to obtain visualization of the robot's current state.
• We relocate the PyBullet built-in simulated camera to point to the current robot location, then return the RGB pixel array values directly obtained from the PyBullet camera.
• We can use the *pyplot.show()* function from the Matplotlib library to visualize the RGB array.

disconnect():
• Used to disconnect from the PyBullet physics simulation engine.

Reflection:
During this project, we have deepened our understanding of Gym and PyBullet. To simplify our problem, we made several adjustments. Despite the original Aliengo robot model having a total of 16 rotary joints, we chose to utilize only the upper and lower limb joints of all four legs, freezing the hip, toe and other irrelevant joints, to avoid unnecessary skewness. We achieved this by editing their URDF files. Additionally, although the Aliengo's joints can handle continuous values, with velocity up to 26.5 rad/s and torque up to 40Nm, we only allow +1, -1 rad/s velocity, and maximum torque of 1, 10 N/m. This is to discretize and reduce our action space. In the future, we might adjust our action space (expand to continuous), and make suitable changes to our state & observation data type, to better suit the actor-critic RL algorithms such as deep deterministic policy gradients (DDPG), which usage is targetted for continuous spaces. The rewards’ values and their calculation may also require further adjustments to enhance training efficiency.